ARTIFICIAL INTELLIGENCE LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AIM 692 September 1982

POLICY-PROTOCOL INTERACTION IN COMPOSITE PROCESSES

by

C.J. Barter

Abstract

Message policy is defined to be the description of the disposition of messages of a single type, when received by a group of processes. Group policy applies to all the processes of a group, but for a single message type. It is proposed that group policy be specified in an expression which is separate from the code of the processes of the group, and in a separate notation. As a result, it is possible to write policy expressions which are independent of process state variables, and as well use a simpler control notation based on regular expressions. Input protocol, on the other hand, applies to single processes (or a group as a whole) for all message types. Encapsulation of processes is presented with an unusual emphasis on the transactions and resources which associate with an encapsulated process rather than the state space of the process environment. This is due to the notion of encapsulation without shared variables, and to the association between group policies, message sequences and transactions.

This research was done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial-intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

1. Introduction

The important aspects of concurrent language design are communications, synchronisation and composition of processes. The first two have been extensively studied, focusing on questions such as control, scheduling and nondeterminism, and problems such as deadlock, starvation and fairness. Less has been said about how complex processes may be composed from other processes, and ultimately from elementary sequential operations and communications primitives. This paper discusses group composition techniques, and the communications interfaces between processes when they are organised as a group.

When a message is sent to a group of processes as a whole, one or more of the component processes may receive it. We distinguish two aspects of group message reception in systems where messages are typed. Firstly, processes are typically provided with the means to select messages for reception, by scheduling arrangements such as system queues, or by user code involving local variables to choose between messages of different types. We define **group input protocol** to be the input behaviour of the group as a whole, for *all* message types. Secondly, we define for *each* message type, a **group policy** which determines the disposition of messages within the group.

We shall argue that policies have more to do with the **transactions** handled by groups than the reception of individual messages by processes, and are consequently better expressed at the group level. Because the control of group policy will be predicated on transaction attributes rather than process variables, and because the control issues seem simpler, a separate notation is proposed for group policy. The notation also provides for the encapsulation of one process by another without the use of shared variables.

2. Processes and Modularity

Language proposals for concurrent systems usually define a basic component, an asynchronous process with facilities for external communications and synchronisation. The process is basic in the sense that it is the building module of concurrent systems. The details of the proposals vary a great deal, and we shall mention some which have an influence on the way processes may be composed together.

One difference is whether communications is mainly by access to shared memory, or by message passing. In shared memory systems (Simula67 [Dahl 70], Monitors [Hoare 74], Concurrent Pascal [Brinch Hansen 77], Modula [Wirth 77]), processes communicate by writing and reading shared variables. Access to shared objects gives a tight coupling of processes and can result in efficient implementations. Synchronisation can also be achieved by setting and testing shared variables, either by ordinary assignment or through

special signalling facilities of the language.

The early proposals which eschewed shared memory (PLITS [Feldman 79], Communicating Sequential Processes (CSP) [Hoare 78], Distributed Processes (DP) [Brinch Hansen 78], Actors [Hewitt 77]) promoted message passing in various forms on the grounds of simplicity, reliability and clarity of expression, at least with respect to communications and synchronisation. It is interesting to note that some of the most recent proposals (Synchronising Resources (SR)[Andrews 81], E-CLU [Liskov and Scheifler 81], Modular Processes (MP) [Choi 81]) allow shared variables (with the recommendation that they be used sparingly and with care). The sharing of variables occurs within an explicit grouping of processes (viz. the resource in SR, guardians in E-CLU and the node in MP).

Communications and synchronisation issues are often difficult to separate in particular language proposals, for it is frequently the case that both aspects are involved in the same language feature: For example, the input and output commands of CSP are the sole means of communication *and* synchronisation. These issues have been neatly separated by Choi [Choi 81], where for each communication event there is a process which provides a service, and a process which is requesting a service (the sender of the message). Synchronisation is generally the concern of the sender of the message, and there are three possible arrangements, the **no-wait send**, the **wait send** and the **remote call**. With no-wait send, the sending process does not synchronise with the destination process, and continues execution after sending the message (e.g. PLITS). With wait send, the sending process synchronises with the *receipt* of the message by the destination process, then both processes continue independently (e.g. CSP). With remote call, the sending process synchronises with the *completion* of the service requested by the sender and invoked by the receipt of the message (e.g. DP).

From the point of view of the receiver, three kinds of service are identified, **message service, procedure service** and **subprocess service**. A message service simply receives the message, perhaps assigning values to local variables in the receiver, and the receiving process then continues normal execution. If the message requires a reply, it must be explicitly constructed and sent by the receiver as a new communication (e.g. CSP and PLITS). With procedure service, message reception invokes a procedure to handle the message, which may also construct the value of a reply (the "out" variables in DP and Ada [Ichbiah 79]). Lastly, a service may be provided by a process rather than a procedure, for greater concurrency. In MP, subprocesses are created dynamically to handle subprocess service requests, while in SR, all requests are handled by processes, but the processes are not dynamically created. The arrangements for sending and receiving described above are largely orthogonal, and all meaningful combinations have been proposed in the literature. The proposals for grouping processes advanced in this paper permit the construction of process groups which achieve all the arrangements for sending and receiving surveyed

above.

Communications is mediated by arrangements such as sender-receiver pairing (as in CSP), ports [Balzer 71], message types [Milne and Milner 79], transactions [Feldman 79], constructions [Barter 78], pattern matching [Hewitt 79] and various notations such as Path Expressions [Campbell and Haberman 74], and Input Tools [van den Bos et al 81].

Major differences exist in the structure of the processes themselves, largely determined by the kinds of service provided. In CSP, the basic process is simply a list of sequential commands, using a nondeterministic guarded command notation to control input, output and ordinary sequential execution. The communications commands appear as in-line code. In contrast, DP provides a process with service procedures which may be called remotely, using a monitor-like discipline. A process may have a conventional process body as well, and the execution of the main body and the service procedures interleave in an unusual way [Welsh et al 80]. Ada has both in-line message receivers (entries) and communications procedures, in an attempt to combine the advantages of CSP and DP. The proposals for grouping processes in this paper are independent of process structure; the example program at the end of the paper uses in-line code for services, but it is easy to see how the other kinds of service may be used.

2.2 The Composition of Systems of Processes

A simple way to compose processes is to form a loose grouping of processes within a common communications environment, with a global convention for process names and messages. Various refinements of this model have been proposed which provide ways of restricting the scope of these names. For example, Milne and Milner use an operator to restrict the visibility of port names [Milne and Milner 79]. CSP uses textual nesting of processes (parallel commands) and Algol-like scope rules for access to variables in different processes. Thus there are shared variables, but a "disjointness" property ensures that there is no shared write access.

Textual nesting has also been used to construct hierarchical groups of processes with scope rules on process names to hide the process structure of groups; from the point of view of the sender of a message, the destination is simply a process. The destination may in fact be a group of processes, and the primitive process within the group which receives the message is determined by the group composition and the type of the message [Barter 78]. Structure hiding has been achieved in CSP by the use of a "hole-in-scope" rule whereby a process name is known in all of the enclosing processes but not in the named process itself; structure hiding is used in a stepwise refinement programming methodology, where each refinement step adds an additional process to a group in order to modify the group

behaviour [Hoare and McKeag 79]. Shapiro has developed this methodology through an extension of CSP which adds some flexibility to the naming conventions for processes and message constructors, and applied it to a large system design [Shapiro 80].

Some recent proposals (SR, E-CLU, MP), influenced by the additional considerations of distributed systems, have defined a middle-level structure involving a group of processes, and some shared objects (usually variables). This grouping may be regarded as the counterpart of a processor node in a network of processors. The authors of SR and MP regard these special groups as being different to processes, and do not allow arbitrary nesting of processes and groups.

The most interesting composition ideas have come from languages which were not primarily intended for concurrent programming, but had a strong object-oriented approach and with particular applications in mind (Simula67, Smalltalk [Kay and Goldberg 77, Ingalls 78], Thinglab [Borning 81] and Lisp Machine Flavors [Cannon 79, Weinreb and Moon 81]). The reason is that without the complication of concurrency it is natural to exploit the advantages of shared memory, and this has been done in most imaginative ways. In these languages we see the composition of processes to mean the actual merging of state spaces, process bodies and service procedures, involving a much tighter composition than the loose coupling described earlier. Simula67 introduced the idea of **class concatenation**, where a class could inherit the attributes of another class. By this method, superclass hierarchies could be constructed. The original intention was to provide language support for program modularity, where the modules (classes) would correspond closely with the conceptual layers of a system design. Class concatenation also foreshadowed another important kind of group composition where one object encapsulates another (see later). The idea of class introduced by Simula67 has been extraordinarily influential, even though some of its details have been criticized (the details of concatenation, Algol scoping and remote accessing of class attributes).

2.3 Superclass Schemes and Process Composition

Languages such as Simula67 and Smalltalk allowed class objects to inherit attributes (procedures, methods and even variables) from other classes by class concatenation. However, the structures which can be built this way are strictly hierachical, and may be classified as single superclass systems. Multiple superclass systems such as Thinglab and Flavors allow inheritance lattices. The inheritance mainly applies to the inheritance of methods (which may be viewed as message services), although there may be some state space sharing as well.

In the Flavor system, a flavor (a class-like specification) can be constructed from other

flavors by a technique called "mixing". A mixed flavor may have components such that more than one component has a method with the same name; an important contribution of the Flavor system is that if an object of the mixed flavor is instantiated, and a method of that object is invoked, *more than one method may be executed* from the set of component methods. The programmer selects a one of a set of **method combinations** to control which component methods are executed, and in which order. The default method combination is called **daemon combination** which allows methods to be classified as **before, primary** or **after** methods; all before methods are handled first, then the single primary method, and finally the after methods are handled. Within the before and after groups of methods, method order is determined by the order in which component flavors are mixed to form the composite flavor (in fact a tree walk order). In every case, the message handling policy is statically determined by the text of the flavors and methods. Our proposal differs in several ways. Firstly, the specification of group policy is separated from that of group composition; secondly, policy is expressed only at the group level, and not within methods, and finally, dynamic policies will be allowed (dynamic in the sense that method ordering can change depending on the execution environment).

3. Communications Policy

In this section we address a question which is fundamental to any proposal for forming processes into groups, namely how is a message received within a group when it is sent to the group as a whole? This question may be simplified by using message types and ensuring that there is always exactly one process in the group able to receive messages of that type. We define **group policy** to be a specification of how messages of a given type will be received within the group, and this will be the key concept upon which other ideas concerning transaction handling and encapsulation will be based. We shall now examine more flexible policies such as broadcasting to all processes able to receive the message, or the selection of some subset of those eligble. Of course policy may be implemented in an additional "policy manager" process (dispatcher) associated with the group, but we shall describe policies in a descriptive notation through **policy expressions**, examples of which now follow.

Consider a group of processes P, and a message type "msg". Let (P1, P2, ... Pn) be those processes of P which accept messages of type msg. Three basic policies are now given by example.

- A policy of **selection** for P is written: **policy** msg:(P1[] P2)

Only processes P1 and P2 are considered as possible destinations for messages of type msg. The choice between P1 and P2 is nondeterministic, all other things being equal. (An implementation could choose the first process ready to receive.) For example, consider a print request sent to a pool of print resources, and the request may be satisfied by any member of a subset of printing resources (e.g. those three which are nearby). The policy for the group "printer-pool" may be expressed:

policy print-request: (print-resource(1) [] print-resource(2) [] print-resource(3))

- A policy of **broadcasting** for P is written: **policy** msg:(P1 // P2)

  Both P1 and P2 receive the message, but the order is unspecified. For example, a request for some services may also be logged on an accounting file, and registered with a load monitor. The policy for such an encapsulated printer pool may be expressed:

  policy print-request: (printer-pool // accounts // load-monitor)

- A policy of **serial broadcasting** for P is written: **policy** msg:(P1 ; P2)

  Both P1 and P2 receive the message, but process P1 must complete the processing of the message before P2 starts. Serial broadcasting is likely to be most useful in groups with shared memory; for example, it is the default policy for calling combined methods in the Lisp Machine Flavor system. Both forms of broadcasting require a convention when used with remote call, to determine which service sends the reply; see later for default policies.

An important degenerate case is **policy** msg:(P1), which simply directs all messages of type msg to P1.

A policy expression describes the disposition of *every* message received by the group, and therefore may be regarded as a repeating construct. (Additional notation will be introduced later to specify repetition of inner components). A policy expression for a group cannot directly affect the reception of messages by that group; policy only determines the disposition of a message when it is received by the group.

Compound policy expressions may be formed in three obvious ways:

o By nesting groups as in:

policy for group P is **policy** msg:(P1 [] P2)
policy for group Q is **policy** msg:(Q1 [] Q2)
policy for group PQ is **policy** msg:(P // Q)

where a message for group PQ is sent to P1 or P2 and also to Q1 or Q2.

o By expression nesting, e.g. policy for P is **policy** msg:((P1 [] P2) // (P3 [] P4))

where a message for group P is sent to P1 or P2 and also to P3 or P4.

o As a **sequence of policies**, **policy** msg:((P1 [] P2) >> (P3 [] P4))

The initial policy is (P1 [] P2), which directs one message to either P1 or P2. The policy then changes to (P3 [] P4), and after that the policy expression repeats. A sequence of policies achieves a similar effect to **actor replacement** [Hewitt et al 79].

In a language using policy expressions, some convention for default policy would be useful, and perhaps some way of defining message type aliases (a reasonable default would be the selection of a single receiver, using a static criterion such as text order in the group description, or a dynamic selection over all eligible processes).

3.2 Policy Model

The semantics of policies are now given as code for a virtual group message handler. The notation is CSP-like, where "P!msg" is the usual CSP wait send of message "msg" to destination P. The notation is extended so that "P.msg" signifies a remote call to P: if a process Q executes a remote call "P.msg" which activates the guarded command "?msg --> command-list", then "P.msg" in Q does not terminate until "command-list" in P does. Also, the input command "?msg" differs from CSP in that it does not name a sender, but will receive messages of the appropriate type [Barter 78]. The three basic policies are:

policy msg:(P1 [] P2) => [?msg --> [true --> P1.msg [] true --> P2.msg]]

policy msg:(P1 // P2) => [?msg --> [[P1.msg] // [P2.msg]]]

policy msg:(P1 ; P2) => [?msg --> [P1.msg ; P2.msg]]

Note that all the virtual handlers have the same structure, [LHS --> RHS], where LHS is always the virtual input command for the group, and RHS is a simple transformation of the policy expression. Virtual handlers for nested policy expressions are similarly constructed by repeated transformation:

policy msg:((P1 [] P2) // (P3 [] P4)) =>

[?msg --> [[true --> P1.msg [] true --> P2.msg]
// [true --> P1.msg [] true --> P2.msg]]]

Sequences of policies result in sequential composition of virtual handlers; the operator ">>" takes precedence over the others in deriving the virtual handler:

policy msg:(P1 >> P2) => [?msg --> P1.msg] ; [?msg --> P2.msg]

policy msg:((P1 [] P2) >> (P3 [] P4)) =>

[[?msg --> [true --> P1.msg [] true --> P2.msg]] ;
[?msg --> [true --> P1.msg [] true --> P2.msg]]]

4. Communications Protocol

The meaning of a process may be given in terms of its input-output behaviour [Milne and Milner 79]. Behaviour can also be expressed as the set of all possible communication sequences [Hoare 78]. In the Actor model of concurrency, an actor receiving a message may change its local state, send messages to other actors and create new actors. The arrival of a message at an actor is called an event, and local time for an actor is the *arrival ordering* of events. Message sending is not important in the event ordering as the model is asynchronous . However, an event can cause a message to be sent, and hence cause another arrival event; in which case the first event is said to activate the second event. Communications between actors is represented by such *activation orderings*. The meaning of a program is given by the *combined ordering* [Hewitt et al 79, Clinger 81].

In this paper we are interested in control over input messages, and input protocol will mean just the input behaviour of a process. We shall refer to input protocol as **protocol** for short (this is a narrower definition than used in the literature on networks).

The protocol of a process is determined by the mechanisms within the process for selecting the next message to receive from a set of pending messages. These mechanisms depend

upon an ability to discriminate between messages by some global measure such as arrival ordering [Hewitt et al 79], or on the basis of message attributes such as type, sender and priority. Arrival ordering is sometimes used in combination with message attributes as a subsidiary selection criterion (PLITS, COSPOL [Roper and Barter 81]). Four basic mechanisms have been used:

- Firstly, there are processes which have a process body which controls the selection of the next message to be received, using in-line receive commands (the input command of CSP and the entry of Ada). Local variables can be used to control message selection by normal flow of control and by guarding input commands.

- Secondly, there are schemes which have service procedures or processes which are directly accessible to other processes, without the control of a "main body" (e.g. SR, E-CLU). Local variables can be used to guard access to services.

- Thirdly, message reception can be entirely determined by arrival ordering; in Actor languages, messages can be typed (implicitly by pattern), and the pattern may be matched against a set of alternative actions, but the pattern matching determines the body which is to be executed, not the message to be selected. While languages such as CSP have "choice nondeterminism" which affects message selection ordering, Actor languages only have "arrival nondeterminism" due to asynchronous communication [Clinger 81].

- Finally, there are languages which use a separate notation to control message selection, such as Path Expressions [Campbell and Haberman 74] and Input Tools [van den Bos et al 81].

Path Expressions are based on regular expressions, using the names of the service procedures of a resource. As well as scheduling service requests, Path Expressions also control the amount of concurrency in the resource services.

The Input Tool Process model provides an event-driven model based on input events, controlled by **input rules**. An Input Tool has a name, an input rule, a tool body and an initialisation section. Tools may be composed in parallel, or nested. An input rule is based on a regular expression notation, using the names of other tools. If an input rule is matched, the tool body is executed, and that tool name may cause further matching in an in input rule at a higher level. Direct communications between processes involves a match between a **send** command in one process and a **receive rule** in another (a tool may specify a **receive rule** instead of an input rule). A parser uses the input rules to dynamically construct the currently "active" structure of input tools (a tree for each proces, whose terminal nodes

are basic tools with receive rules). Inputs which do not match the current structure are ignored. Thus input rules control input protocol and, as we shall show, some aspects of what we have called policy.

Input rules can be used to control both policy and protocol (indeed the authors do not draw the distinction). Because the Input Tool model has strong similarities with our policy proposals and strong differences with our treatment of input protocol, we shall discuss the model in some detail:

4.2 The Input Tool Model

The example of a printer server is given [van den Bos et al 81]:

```
tool printer = input (first-line; |more|: source --> line$)$ end
  bool more; process set source;
  tool first-line = input line end
        if more
        then source := {sender}
        fi
  end
  tool line = receive string msg;
        more := (msg <> EOF);
        if more
        then lineprint(msg)
        else skip-page
        fi
  end
end
```

The input rule uses ";" for sequences of matches, "$" for repetition, and |<boolean-expression>|: for guarding. The notation "source -->" restricts input messages to be from a particular sender, in this example it is the one bound by the assignment "source := {sender}".

When the tool "printer" is activated, the parser activates the tool "first-line", and through it, the tool "line"; "line" is a basic tool which receives a message "msg", which matches its receive rule, and so the body of "line" is executed. This matches the input rule of "first-line", and so its body is executed, and the component "first-line" of the top-level is matched. The parser now moves to the next component of the top-level input rule: this will be "|more|: source --> line$" if the boolean guard "more" is true, but if "more" is false

that component will be invisible to the parser, and so the next component will again be "first-line".

A second example shows input rules used to direct a message of the same type to alternative tools:

```
tool squash = input |go-on|: (star + nostar)$ end
        ...
  tool star = input character(c): |c = "*"| end
        ...
  end
  tool nostar = input character(c): |c <> "*"| end
        ...
  if c = EOF then go-on := false fi
  end
        ...
init ... ; go-on := true end
end
```

The operator "+" specifies a choice between two tools, and the input rule 'character(c): |c == "*"|' uses a post-test on the value of the parameter "c", so that the post-test must succeed if the rule is to match.

An example of a bounded buffer is given to illustrate an input rule controlling a simple input protocol; the example is given here in abbreviated form:

```
tool buffer = input (|count ≤ size|: put + |count > 0|: get)$ end
        ...
  tool put = receive char c;
        ...
  end
  tool get = receive;
        ...
  end
        ...
end
```

The parser does not activate the tool "put" if the buffer is full, and similarly does not activate the tool "get" if the buffer is empty. The boolean guards are computed within the bodies of put and get.

The example programs show three uses of input rules. The first example shows an input rule specifying a message policy: an input line is processed by either one or both tools. The purpose is to provide some encapsulation of tool "line" by tool "first-line". This particular encapsulation does not generalise well; encapsulation will be treated later. The third example also specifies policy for input characters, using the tool "star" or "nostar" depending on the data. In both examples, the input rules affect policy but not input protocol. In the second example, the input rule controls protocol in the sense that it directly determines the scheduling of input requests. In all three examples the input rules exercise control through shared variables.

The use of program variables in these expressions allows arbitrary interactions between the expressions and the code of the processes controlled. But typical protocol and scheduling descriptions do involve variables which are local to (and sometimes shared between) the processes concerned; this is a strong reason not to place these descriptions in a separate expression, but to leave them in the code of the processes themselves. On the other hand, we shall show that policies have less to do with individual processes and their variables, and more to do with groups of processes and message sequences; for this reason we shall argue that group policy is better placed in a separate description associated with the group, and that a separate notation is useful for its description.

Because the method of process composition suggested in this paper does not involve message re-scheduling, the protocol of a group is simply the merge of individual protocols (i.e. all orderings which preserve the partial ordering of the component processes). Next we show how policy and protocol may interact without using shared variables in either the processes or the policy expressions.

5. Policy-Protocol Interaction

Consider a group of children and gifts arriving.

> The group is: (Sharon, Carol, Jenny, Michael)
> The messages are: (gift, boy-gift, girl-gift)

Some example policies are:

> **policy** gift:(Sharon >> Carol >> Jenny >> Michael) -- i.e. take turns.
> **policy** girl-gift:(Sharon [] Carol [] Jenny) -- i.e. choice
> **policy** boy-gift:(Michael) -- i.e. single receiver

The three policies are independent - e.g. the policy for messages of type "gift" has no

influence on the policy for messages of the type "girl-gift".

Consider the following policies:

policy gift:(Sharon // Carol // Jenny // Michael)
policy gift:(Sharon ; Carol ; Jenny ; Michael)

In these two policies, for every incoming message of type "gift", four messages are copied to the group members, concurrently in the former policy and sequentially in the latter.

The policy for "girl-gift" is not fully determined by the policy expression; an implementation may have some additional criteria for making the choice, such as choosing the process which has been waiting longest for a message of that type. (An alternative strategy is to choose without consideration of whether processes are ready or not, and wait if the chosen process is not ready; this can lead to more deadlocks than the first strategy). Rather than regarding the previous as an implementation issue, the selection method could be part of the language definition and exploited to schedule message reception or synchronisation; but this encourages a dangerous interdependence between processes in a way which undermines modularity and clean interfacing. We now examine some policy-protocol interactions which depend only on more general aspects of communications:

- A process may **terminate**, which is a most drastic change of protocol. The most desirable behaviour with respect to group policy if a component of the group terminates will depend on the composition method. If the terminated component is composed with the policy operators "//" or "[]", then the process may be dropped (dynamically) from the policy provided that there is some live component to receive the message; if not, the group should **abort**.

- A process may close a typed message service, which is similar to termination, but only with respect to that message type and the corresponding message policy.

- The policy for a group is by definition a repeating construct, and as such associates with message sequences rather than a single message. The policies given earlier could have made this explicit with a repetition operator such as the Kleene star, e.g.: **policy** msg:(P1 [] P2)*.

  An explicit operator is necessary to express repetitions of policies *within* sequences of policies, e.g.:

policy msg:((P1 [] P2)* >> (P3 [] P4)*)

In this policy, a sequence of messages is dispatched under the policy (P1 [] P2)*, before the policy changes to (P3 [] P4)*. Some means of breaking the sequence is required, and we propose an explicit **break-policy** signal rather than a test on a program variable. A logically associated sequence of messages is usually called a **transaction.** It is useful to strengthen the attributes of a transaction by sender-receiver bindings, and two operations are proposed for this purpose: **attach-sender** and **break-sender.**

break-policy has the following effect: If the current policy is part of a sequence of policies, and not the last policy in that sequence, the next policy becomes the current policy; otherwise the break is passed up to the next level, if any. When there is no "next level up" (the group is not a component of another group), the policy at that level does not signal a break, but restarts the entire policy expression at that level. (Repetition in policy expressions and the **break-policy** operation are similar to **catch** and **throw** in some versions of Lisp [Weinreb and Moon 81].)

attach-sender restricts all further messages received by the group to be from the sender of the last message, and this prevails until **break-sender** is executed within a process of the same group.

The three policy operations described above will be illustrated in an example after a discussion of **encapsulation.**

6. Encapsulation

Simula67 supports a form of encapsulation through class concatenation; a special symbol **inner** is used to mark a point in the code of the body of a process, to identify where the code body of the encapsulated may be regarded to notionally execute. A similar encapsulation facility with respect to method bodies is available in the Flavor system (wrappers).

Hewitt's serialisers/guardians may be used to encapsulate a resource process by intercepting and re-scheduling all communications with the resource. The guardian acts on behalf of the user of the resource. The purpose of the encapsulation is to enforce a stronger protocol than that of the resource itself; i.e. the resource may have been designed without considering the possibility of careless or malicious use, and the encapsulation is then designed to compensate for this.

Although shared variables are often exploited to provide the kinds of run-time environment encapsulation possible in the languages described above, we shall only discuss sharing through the communication environment, rather than through process state spaces. The most important communication attributes to be shared are those to do with transactions involving more than one message passing event. Examples of transaction attributes of interest such as policy-sequence bindings and sender-receiver bindings have already been mentioned.

We now introduce the construct **inner** to provide some encapsulation abilities in groups. The name **inner** is borrowed from Simula67, but because it is used without access to shared variables, its semantics is different to that in Simula. A process receiving a message will execute its command list (or service) up to the occurrence of the **inner** marker, and skip the remainder; when an entire policy expression is complete, all command lists whose remainder parts were skipped are then executed, in reverse order to the order in which they were skipped. Each remainder will be executed as many times as it was skipped in each component of the policy expression.

Thus an encapsulating process encapsulates the transactions or message sequences of the encapsulated process, rather than its execution environment; but this is often what is required anyway. A common use of encapsulation is resource locking, where only requests of the current transaction are allowed to access the resource, and all other requests are locked out for the duration of the transaction. To achieve this effect, encapsulating process could contain the following code:

```
... lock; inner; unlock; ...
```

In the following example, **inner** is used to illustrate head and tail encapsulation in the printer problem. The example is an extended version of the earlier printer example, with the added requirements that each file be printed with a header and a trailer, both containing information extracted from the first line of the file, and that empty files should not cause a page-skip. The programming language used is the same as that used for the description of virtual group policy handlers [Barter 78] for the sake of example, and it is not intended that the group policy model associate with any specific language.

```
[ Printer ::
    group-members : (Newfile, Printlines, ...)
    policy line : (Newfile* >> Printlines*)
    ...
    .. policies for other message types
    ...
```

```
]
//
[ Newfile ::
   *[ ?line -->
      [ line.eof --> skip
      []not line.eof -->
         attach-sender;
         print-header(line);
         break-policy
           inner;
         print-trailer(line);
         page-skip;
         break-sender;

]  ]  ]
//
[ Printlines ::
   *[ ?line -->
      [ line.eof --> break-policy
      []not line.eof --> print(line)
]  ]  ]
```

The modularity achieved is typically that which is to be expected from careful encapsulation. The process Newfile only performs operations at the file level, either empty ones which are skipped, or non-empty ones which have headers, bodies (for which **inner** is a surrogate), and trailers. The process Printlines just handles sequences of lines under some prevailing policy, breaking at end-of-file.

Note that the sender-receiver binding is now handled in the same process, and that the header and trailer procedures use the same value of line as their parameters.

Conclusions

Message policy has been defined to be the description of the disposition of messages of the same type, when received by a group of processes. Group policy applies to all the processes of a group, but for a single message type. It is proposed that group policy be specified in an expression which is separate from the code of the processes of the group, and in a separate notation. Separate specification seems natural, for policies are associated with transactions and message sequences rather than the details of processes; for this reason it is possible to write policy expression which are independent of process state variables,

and as well use a simpler control notation based on regular expressions.

Input protocol, on the other hand, applies to single processes (or a group as a whole) for all message types. When policy aspects are separated from input protocol, scheduling is what usually remains, and scheduling often has strong associations with process state variables; for this reason it is often difficult to specify protocol expressions without using control constructs which access process state variables. Accordingly, we leave control over protocol in the code of the processes themselves.

Encapsulation of processes is presented with an unusual emphasis on the transactions and resources which associate with an encapsulated process rather than the state space of the process environment. This is due to the notion of encapsulation without shared variables, and to the association between group policies, message sequences and transactions.

We have tried to avoid committment to any particular language within the general message-passing group surveyed, though there are important interactions which will affect group composition and policy expression, as well as implementation (e.g. the prescence of remote call in a language will significantly influence implementation strategies). We have not argued against shared variables (in small amount), but have shown what is possible without them. The example program given used a CSP-like syntax, and suggested a load-and-go execution environment. We believe that the ideas transfer to incremental execution environments as well, such as provided by Lisp. This could be done in several ways. Firstly, policies could be expressed as Lisp (Lisp Machine) functions, dispatching messages to objects of the appropriate flavor and wrappers; the programmer would have to enumerate all the flavor mixes required by the policies. This achieves dynamic control over over method execution by object replication. A macro technique could make this easier to use. Finally, the flavor and wrapper concepts could be unified, and generalised so that the policy for executing methods could be controlled dynamically, rather than being tied to the order in which flavors are combined.

Two significant problems need immediate consideration. Firstly, we have discussed the formation of groups from classes rather than objects, and the difference is important in languages with dynamic process creation. Secondly, we have not examined the question of objects being components of more than one group (shared objects).

Acknowledgements

Marc Shapiro was an invaluable colleague in the early stages of this work, and contributed the name "group policy" to the notion of controlling the disposition of messages within a group. Mike Brady read many drafts, and tried to steer me down the path of rigor with a

mixture of criticism and encouragement. Thanks to Howard Cannon for Flavors and the Lisp Machine windows which vindicate them, and to Richard Stallman for discussions which suggest that the ideas presented here may be most interesting in a Lisp environment. Thanks are also due to William Kornfeld, Maurice Herlihy and Pierre Lescanne.

References

Andrews, G.R., "Synchronizing resources". ACM Trans. on Prog. Lang. and Sys. 3, 4 (Oct. 1981), 405-430.

Balzer, R.M., "Ports - a method for dynamic interprogram communication and job control". Proc AFIPS Spring Joint Comp. Conf., 19, (1971), 485-489.

Barter, C.J., "Communications between sequential processes". TR 34, Dept. Comp. Sci., University of Rochester, 1978.

Borning, A., "The programming language aspects of Thinglab, a constraint-oriented simulation laboratory". ACM Trans. on Prog. Lang. and Sys. 3, 4 (Oct. 1981), 353-387.

van den Bos, J., Plasmeijer, R. and Stroet, J., "Process communication based on input specifications". ACM Trans. on Prog. Lang. and Sys. 3, 3 (July 1981), 224-250.

Brinch Hansen, P., "Distributed processes". Comm ACM 21, 11 (Nov. 1978), 934-941.

Brinch Hansen, P., "The programming language Concurrent Pascal". IEEE Trans. Software Eng. 1, 2 (June 1975), 199-207.

Campbell, R.H. and Habermann, A.N., "The specification of process synchronisation by path expressions". Lecture Notes in Computer Science, 16, (1974), Springer-Verlag.

Cannon, H.I., "Flavors - a non-hierarchical approach to object-oriented programming". Report associated with the MIT Lisp Machine project, 1979.

Choi, Y.J., "Process interaction in Modular Processes". TR 81-06, Dept. Comp. Sci., Univ. of Adelaide, 1981.

Clinger, W.D., "Foundations of Actor semantics". AI-TR-633, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1981.

Dahl, O-J, Myrhaug, B. and Nygaard, K., "The Simula67 common base language". Norwegian Computing Center, Norway, 1970.

Feldman, J.A., "High level programming for distributed computing". Comm ACM 22, 6 (June 1979), 353-368.

Hewitt, C., "Viewing control structures as patterns of passing messages". Artificial

Intelligence, 8, 1977, 323-363.

Hewitt, C., Attardi, G. and Lieberman, H., "Specifying and proving properties of guardians for distributed systems". In Semantics of Concurrent Computation, Lecture Notes in Computer Science, 70, (1979), Springer-Verlag.

Hoare, C.A.R., "Monitors: an operating system structuring concept". Comm ACM 17, 10 (Oct. 1974), 549-557.

Hoare, C.A.R., "Communicating sequential processes". Comm ACM 21, 8 (Aug. 1978), 666-677.

Hoare, C.A.R., "Some properties of predicate transformers". J.ACM 25,(July 1978), 666-677.

Hoare, C.A.R. and McKeag, R.M., "The structure of an operating system". Ecole de l'INRIA, Methodologie de la programmation, Mar., 1979.

Ichbiah, J., et al, "Preliminary Ada reference manual". SIGPLAN Notices, 14, 6, (June 1979).

Ingalls, D.H.H., "The Smalltalk-76 programming system: design and implementation". Conf. Rec., 5th Ann. ACM Symp. Principles of Programming Languages, Tuscon, Ariz., (Jan. 1978), 9-16

Kay, A. and Goldberg, A., "Personal dynamic media". Computer 10, 3 (March 1977), 31-42.

Liskov, B. and Scheifler, R., "Guardians and actions: linguistic support for robust, distributed programs". CSG Memo 210, (1981) Laboratory for Computer Science, MIT.

Milne, G.J. and Milner, R., "Concurrent processes and their syntax". J.ACM 26, (1979), 302-321.

Roper, T.J. and Barter, C.J., "A communicating sequential process language and implementation". Software - Practice and Experience, 11, 11 (Nov. 1981), 1215-1234.

Shapiro, M., "Une methode de conception progressive des systemes paralleles, utilisant le langage CSP". These de Docteur-Ingenieur, Sept. 1980, INP-ENSEEIHT, Toulouse, France.

Svobodova, L., Liskov, B. and Clark, D., "Distributed computer systems structure and semantics". Laboratory for Computer Science TR-215, (1979), MIT.

Weinreb, D. and Moon, D., "Lisp Machine Manual". 4th edition, 1981, Artificial Intelligence Laboratory, MIT.

Welsh, J., Lister, A. and Salzman, E.J., "A comparison of two notations for process communication". In Language Design and Programming Methodology, Lecture Notes in Computer Science, 79, (1980), Springer-Verlag.

Wirth,N., "Modula: a programming language for modular multiprogramming". Software - Practice and Experience 7, 1 (Jan. 1977), 3-35

CJB/cjb